Lista zadań nr 9

Rozważ następujący język wyrażeń arytmetycznych, ewaluator, wyrażenia w odwrotnej notacji polskiej (ONP), kompilator do ONP i maszynę wirtualną ze stosem obliczającą wartość wyrażeń w ORP:

```
type expr =
  | Int of int
  | Add of expr * expr
  | Mult of expr * expr

let rec eval (e : expr) : int =
  match e with
  | Int n -> n
  | Add (e1, e2) -> eval e1 + eval e2
  | Mult (e1, e2) -> eval e1 * eval e2

type rpn_cmd =
  | Push of int
  | RAdd
  | RMult

type rpn = rpn_cmd list

let rec to_rpn (e : expr) : rpn =
  match e with
  | Int n -> [Push n]
  | Add (e1, e2) -> to_rpn e1 @ to_rpn e2 @ [RAdd]
  | Mult (e1, e2) -> to_rpn e1 @ to_rpn e2 @ [RMult]

let rec eval_rpn (r : rpn) (s : int list) : int =
  match r, s with
  | [], [n] -> n
  | Push n :: r', _ -> eval_rpn r' (n :: s)
  | RAdd :: r', n1 :: n2 :: s' -> eval_rpn r' (n2 + n1 :: s')
  | RMult :: r', n1 :: n2 :: s' -> eval_rpn r' (n2 * n1 :: s')
  | _,_ -> failwith "error!"
```

Zadanie 1. (2 pkt)

Udowodnij poprawność kompilacji do ONP, czyli że dla dowolnego wyrażenia `e : expr` zachodzi

```
eval_rpn (to_rpn e) [] = eval e
```

Wskazówka: Próbując udowodnić to twierdzenie bezpośrednio przez indukcję względem struktury `e` szybko można napotkać na przeszkodę: założenie indukcyjne jest za słabe. Żeby pokonać tę przeszkodę, trzeba najpierw udowodnić twierdzenie bardziej ogólne i wywnioskować z niego twierdzenie, które tak naprawdę chcemy udowodnić. Jakie twierdzenie bardziej ogólne udowodnić? Proszę spojrzeć na za słabe twierdzenie indukcyjne i zobaczyć, czego w nim brakuje (prawdopodobnie chcemy mówić o stosach innych niż pusty, no i gdzieś pewnie pojawi się `@`).

Zadanie 2. (2 pkt)

Zaimplementuj funkcję

```
from_rpn r : rpn -> expr
```

kompilującą wyrażenia w ONP do składni abstrakcyjnej wyrażeń arytmetycznych.

Wskazówka: Kompilator `expr -> rpn` ma strukturę ewaluatora, którego wartościami są wyrażenia w ONP. Podobnie tutaj, kompilator może mieć strukturę interpretera wyrażeń ONP ze stosem, którego wartościami są wyrażenia w składni abstrakcyjnej.

Zadanie 3. (2 pkt)

Zaimplementuj funkcję, która generuje losowe wyrażenie arytmetyczne

```
let random_expr (max_depth : int) : expr = ...
```

gdzie `max_depth` określa maksymalną głębokość drzewa generowanego wyrażenia (przypomnij sobie zadanie 3. z listy 6). Zaimplementuj funkcję

```
let test (max_depth : int) (n : int) : bool = ...
```

która generuje `n` losowych wyrażeń `e` o maksymalnej głębokości `max_depth` i zwraca informację, czy dla każdego z nich zachodzi własność

```
from_rpn (to_rpn e) = e
```

Zadanie 4. (2 pkt)

Zmodyfikuj funkcję `test` z poprzedniego zadania tak, by miała typ

```
test_ce : int -> int -> expr option
```

która zwraca `None`, jeśli wszystkie wygenerowane wyrażenia spełniają własność, albo `Some e`, gdzie `e` jest jakimś wyrażeniem, które nie spełnia własności (kontrprzykładem). Sprawdź działanie tej funkcji modyfikując `from_rpn` lub `to_rpn` tak, by własność nie zachodziła.

Zadanie 5. (2 pkt)

Rozważ typ spłaszczonych programów instrukcji skompilowanego języka FUN, do którego wprwadzamy etykiety typu `string` reprezentowane przez instrukcję `Lbl` oraz skos do etykiety reprezentowany przez instrukcję `Jmp`. Instrukcję trzymające bezpośrednio bloki kodu `ComdJmp` oraz `PushClo` przyjmują teraz etykietę do skoku:

```
module T = struct
type cmd =
  | PushInt of int
  | PushBool of bool
  | Prim of bop
  | Jmp of string
  | CondJmp of string
  | Grab
  | Access of int
  | EndLet
  | PushClo of string
  | Call of string
  | Return
  | Lbl of string
end
```

Zaimplementuj funkcję

```
flatten : cmd list -> T.cmd list
```

która „spłaszcza" program do nowego zestawu instrukcji przez kompilację bloków kodu do etykiet i skoków w odpowiednich miejscach. Do generowania etykiet możesz użyć techniki podobnej do tej z zadania 3. z listy 8.

Zadanie 6. (2 pkt)

Zmodyfikuj maszynę wirtualną tak, by działała z nowym zestawem instrukcji.